ARTIFICIAL INTELLIGENCE LABORATORY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

AIM 691 DECEMBER 1982

OPEN SYSTEMS

by

Carl Hewitt

Peter de Jong

ABSTRACT

This paper describes some problems and opportunities associated with conceptual modeling for the kind of "open systems" we foresee must and will be increasingly recognized as a central line of computer system development. Computer applications will be based on communication between sub-systems which will have been developed separately and independently. Some of the reasons for independent development are the following: competition, different goals and responsibilities, economics, and geographical distribution. We must deal with all the problems that arise from this conceptual disparity of sub-systems which have been independently developed. Sub-systems will be open-ended and incremental--undergoing continual evolution. There are no global objects. The only thing that all the various sub-systems hold in common is the ability to communicate with each other. In this paper we study Open Systems from the viewpoint of Message Passing Semantics, a research programme to explore issues in the semantics of communication in parallel systems such as negotiation, transaction management, problem solving, change, and self-knowledge.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Major support for the research reported in this paper was provided by the System Development Foundation. Major support for other related work in the Artificial Intelligence Laboratory is provided, in part, by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505.

1. Introduction

A goal of conceptual modeling is to aid in the implementation of the next generation of computing systems by applying and unifying results from the fields of programming languages, data bases, and artificial intelligence. A central theme is to have the computer system "understand" not only the environment in which it will operate but the application as well. Our approach to conceptual modeling reflects the authors' views on the way computer systems are developing now and the ways we expect them to evolve. We expect that the standard use of computer applications will involve the interaction of subsystems which have been independently developed at disparate geographical locations [de Jong 80]. In part the lack stems from the fact that research on Open Systems is in its infancy in the basic contributing areas to conceptual modeling (viz. programming languages, data bases, and artificial intelligence). Unfortunately each of these contributing areas has for the most part simply tried to extrapolate the methodology and technology of a single computer system to Open Systems. These extrapolations are inadequate because they fail to treat central problems such as negotiation and self-knowledge. In this essay we discuss how message passing semantics applies to the conceptual modeling of Open Systems. In particular we discuss a number of technologies we are developing to address the problems of modeling Open Systems.

2. Open Systems

The kind of systems we envisage will be open-ended and incremental--undergoing continual evolution. In an open system it becomes very difficult to determine what objects exist at any point in time. For example a query might never finish looking for possible answers. A query to find a bargain priced used refrigerator in good condition would reference information stored in any of a number of personal and organizational computers. Enormous amounts of effort and time could be expended processing the query to find such a refrigerator, without being certain that the best buy has been located. Similarly if a system is asked to find all the telephone numbers of passengers who have ever made reservations on Pan American, it might have a hard time answering. It can give all the telephone numbers it has found so far, but there is no guarantee that another one can't be found by more diligent search. These examples illustrate how the "closed world assumption" is intrinsically contrary to the nature of Open Systems. We understand the "closed world assumption" to be that the information about the world being modeled is complete in the sense that *all and only* the relationships that can possibly hold among objects are those implied by the given information at hand (cf. [Reiter 82]). Systems based on the "closed world assumption" typically assume that they can find all the instances of a concept that exist by searching their local storage. In contrast we desire that subsystems be accountable for having evidence for their beliefs and be explicitly aware of the limits of their knowledge. At first glance it might seem that the closed world assumption, almost universal in the A.I. and database literature, is smart because it provides a ready default answer for any query. Unfortunately the default

answers provided become less realistic as the Open System increases in size.

Much of the communication in Open Systems needs to be on the basis of *dissemination*, which is the electronic analogue of publishing in a magazine or bulk mailed advertising. Once a piece of knowledge is disseminated, other knowledge in the system might be affected. When an airline guide is disseminated, travel plans can include the scheduled flights. When the guide is revised as of a certain date, some travel plans might use the old schedule, and some the new schedule. The distributed data base approach approach does not work very well in Open Systems because of issues of ownership and privacy. For example when a proprietary software system is distributed, the organization distributing the software cannot in general access the copy on the customers premises in order to make updates. Instead revisions and extensions must be disseminated for the customers to integrate into their own systems.

There are no global objects in Open Systems. The only thing that all the various subsystems hold in common is the ability to communicate with each other. We believe that negotiation will be a fundamentally important mode of communication in the operation of Open Systems. Similar concepts and capabilities will develop and evolve in many different locations in Open Systems. For example each bank will offer similar services that differ in detail from the services of its competitors. Systems will need to negotiate the terms and conditions of their transactions. Thus they must have a least a rudimentary language in common-- otherwise there would be no basis for negotiation.

Subsystems need to have self-knowledge in order to function effectively in Open Systems in order to understand its own abilities as well as the limits of its knowledge and power. As knowledge is added incrementally to a subsystem, it must relate the new knowledge to its existing knowledge. Any subsystem can have only partial knowledge of the overall system, and partial power to affect other subsystems. A travel agency has the power to make plane reservations but in general does not have the power to cancel flights. Means for distributed problem solving and negotiation are necessary to combine the knowledge and powers of various subsystems to accomplish application goals. Accommodating the above characteristics of Open Systems will be necessary in order for conceptual modeling to be relevant to the future.

A simple example will clarify some of the above points. The application is a vacation trip. The agents involved are a traveler, a travel agent, and a banker. To make the example more modern, the traveler is at home using a computer workstation to communicate with a travel agency and a bank. The trip will be planned interactively with extensive support from the computer systems. The people in the scenario have their own conceptual models which describe the objects they deal with, the organization and subsystems within which they work, and the applications they can perform. In general each conceptual model has been constructed independently of the others at different times and places. The personal computer system in the home must deal with many travel agents and many banks, each with its own view on how it should run its

own business. These business computer systems are geographically distributed. For the most part they run in a parallel non-stop mode, always ready to interact with their customers. They are united by their ability to communicate with each other. Each workstation user's knowledge is partial: the travel agent does not know about the traveler's dealings with the bank. Each agent's power is limited: the travel agent can arrange the trip, but will not finance it. The model changes with time: the travel agent keeps getting more information on travel possibilities and new trips are continually being planned. Each conceptual modeling system needs to have extensive self-knowledge of its own capabilities and partial knowledge about others: the travel agent uses its self-knowledge to help guide its interactions with customers.

3. Spectrum of Procedures

On the basis of the above discussion, we conclude that interaction with workstation users and negotiation with other systems necessitates the introduction and analysis of problem solving procedures into the conceptual modeling of Open Systems. Algorithmic procedures are ones for which definite properties (e.g. termination, certain outputs always produced, etc.) can be proven to hold (cf. [Lenat 82]). The *stronger* the proven properties, the *more algorithmic* the procedure. In contrast to algorithmic, the measure of coherence of procedures is the extent to which properties are *known* to hold. For example an organization may know that it is able to hire another employee even though it is unable to mathematically prove this. The *stronger* the known properties, the *more coherent* the procedure. In contrast, problem solving procedures are those about which fewer and weaker properties are known. Browsing procedures are even less defined and constrained than problem solving procedures. Managers browse through their organizations looking for problems, successes, and opportunities. New workers browse through their organizations becoming acquainted with the way the organization works. Thus we have a spectrum of possible procedures from browsing to problem solving to coherence to algorithmic:

BROWSING<----->PROBLEM SOLVING <----->COHERENT<----->ALGORITHMIC

Workstation procedures can move both directions along this spectrum. A procedure can be moved in the direction from problem solving to greater coherence by mechanizing more cases, by modifications to ameliorate its shortcomings, etc. A procedure can move from coherence toward problem solving by unanticipated external changes which result in anomalous results, by the discovery that their range of applicability is not as great as previously believed, etc. Movement along the spectrum between browsing and algorithms should be one of the central concerns of the conceptual modeling of Open Systems.

Subsystems in Open Systems have only partial knowledge. The first time a conceptual modeling system encounters the notion of an airline reservation, it will have to be as a problem to be solved. On subsequent occasions when it recognizes a new similar problem, we would like the system to remember the principles

behind how it solved the previous problem and adapt the solution to the new circumstances. It would approach the new problem in a more algorithmic way--generating fewer unsuccessful goals. Open Systems must address the central issue of "coherence" of behavior, i.e. convergence in a set of negotiations [Brady 82]. Starting negotiations is only the beginning; we must develop mechanisms for bringing them to successful conclusions. [Davis, Smith 81]

4. Descriptions

We are developing *Description Systems* to model the relationship of objects in Open Systems [Hewitt, Attardi, Simi 80], [Attardi, Simi 81]. We find it useful to make use of several different kinds of descriptions: atomic descriptions, instance descriptions, etc. For example, FRANCE is an atomic description and (a COUNTRY) an instance description. A user can hypothesize that the two descriptions are related using the following inheritance statement (where we underline key words having a special technical meaning):

```
(FRANCE is (a COUNTRY))
```

which states that FRANCE is a specialization of (a COUNTRY) The import of the above statement is that France inherits all the properties of a country.

Instance descriptions can be specialized using attributions. For example

```
(a COUNTRY (with CAPITAL PARIS))
```

is a specialization of (a COUNTRY). The following expresses the statement that FRANCE is a COUNTRY with capital Paris:

```
(FRANCE is (a COUNTRY (with CAPITAL PARIS)))
```

and hence France inherits all the properties of a country with capital Paris. Multiple partial incremental descriptions form the basis of expressing complicated relationships between objects. For example the system can subsequently be told

```
(FRANCE is (a EURAIL-PASS-MEMBER))
```

which says that France is a specialization of a Eurail Pass Member so that in addition France inherits all the properties of Eurail Pass Members. In this way later descriptions are related to earlier ones in a lattice structure.

Our goal is that *Description Systems* should facilitate the conceptual modeling of the objects and properties of Open Systems. Description Systems should be defined by a rigorous mathematical semantics [Attardi, Simi

81]. The "representation", "frame", "schema", etc. systems developed previously have suffered from a lack of mathematical semantics and from being based instead on sequential list-processing programming languages. Description Systems should have an epistemological basis that is compatible with the realities of Open Systems. Another goal is that Description Systems should be distributed over multiple computers and be able to operate in parallel, thus facilitating communication and sharing in Open Systems. Omega is designed to facilitate the incremental use for the partial description of Open Systems.

5. Actors

To build a conceptual modeling system for Open Systems, we need to develop a coherent understanding of the semantics of concurrent message passing systems. Message Passing Semantics is a research programme to explore issues in the semantics of communication in parallel systems such as negotiation, transaction management, problem solving, change, and self-knowledge. It builds on Actor Theory as a foundation for the conceptual modeling of Open Systems. This involves important aspects of parallelism and serialization beyond the sequential coroutine message passing developed in systems like Simula and SmallTalk.

An *actor system* is composed of abstract objects called *actors*. *Actors are defined by their behavior when they accept communications.* When a communication is accepted an actor can perform the following kinds of actions concurrently: make simple decisions, *create* new actors, *transmit* more communications to its own acquaintances as well as the acquaintances of the communication accepted, and change its behavior for the next message accepted (i.e. change its local state) subject to the constraints of certain laws [Hewitt, Baker 77]. These actions are illustrated in the diagram below which presents a *partial* description of what happened when the actor `ACCOUNT43` with a balance of $10 accepted a request to make a deposit with amount $2 for customer `c2`, and as a result created the actor $12, sent a `Completion` report to `c2`, and became an account with balance $12:

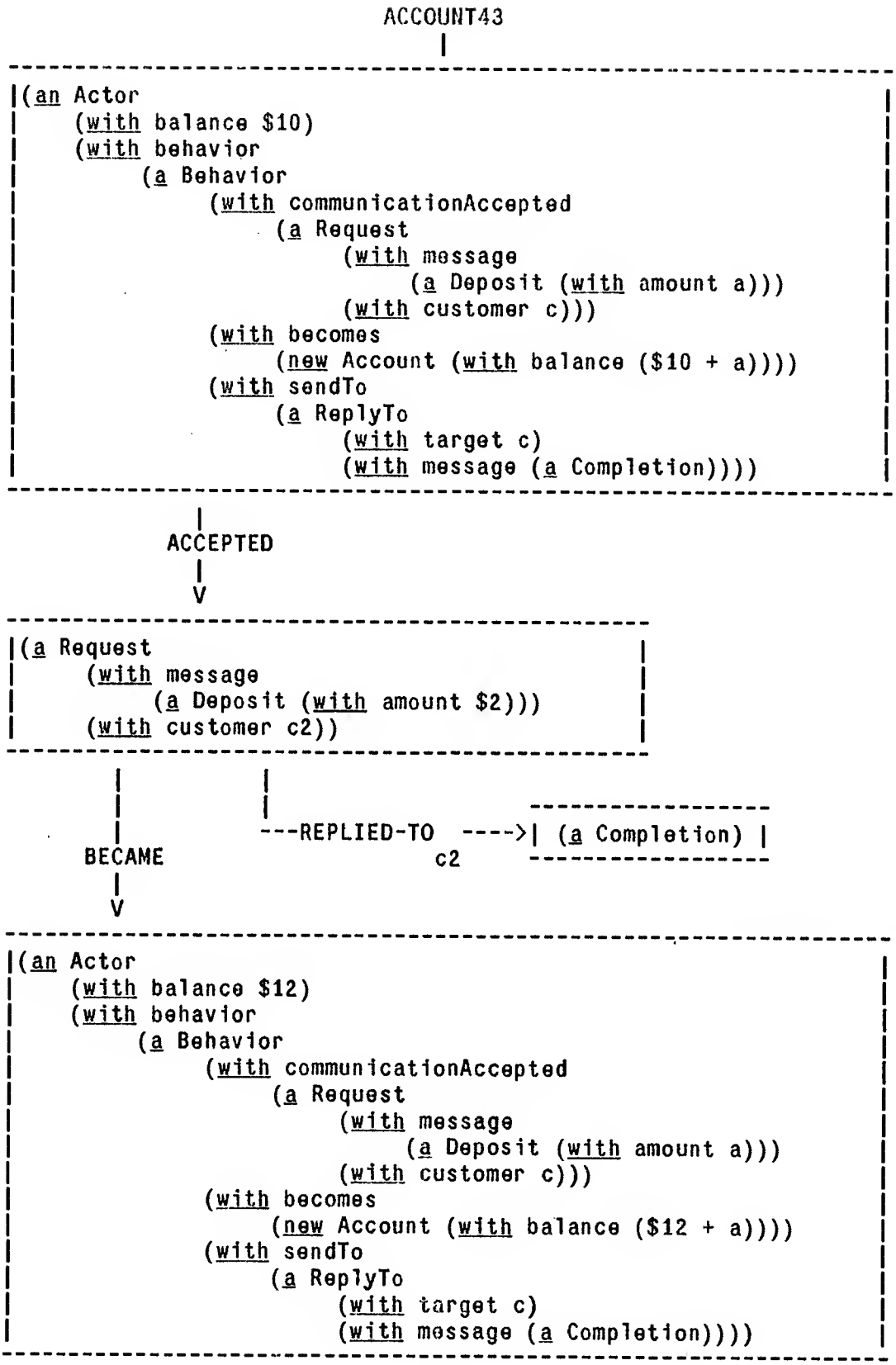

Figure 5-1: A Happening

Control structures such as the one above can be characterized in terms of patterns of message passing [Hewitt 77], [Kerns 80].

The Actor Model addresses the issues of secrecy, protection, and authentication as communication issues. Secrecy and protection are inherently built into actor systems since the only operation that can be performed is to send a communication. There is no way to force an actor to do anything or force it to divulge any information about itself. Authentication can be implemented using communication protocols that use encryption.

The modeling of shared resources is fundamental to Open Systems. In this section we discuss how Message Passing Semantics can help clear up some of the confusion that now surrounds the conceptual modeling of shared resources in Open Systems. Actor systems aim to provide a clean way to implement *effects* (not "side-effects" a pejorative term that has been used as a kind of curse by proponents of purely applicative programming in the lambda calculus). By an *effect* we mean a local state change in a shared actor which causes a change in behavior that is visible to other users. For example sending a deposit to an account shared by multiple users should have the effect of increasing the balance in the account.

6. Concurrent Systems

In this section we present a simple example which illustrates our approach to implementing shared resources in Open Systems. We would like to discuss the conceptual modeling of a `PersonalAccount` resource which is suitable for use by a growing collection of users in an Open System. To a given user, a shared `PersonalAccount` will exhibit indeterminacy in the balance depending on the deposits and withdrawals made by other users. The indeterminacy arises from the indeterminacy in the arrival order of messages at the PersonalAccount.

A `PersonalAccount` is an `Account` with a non-negative balance. A `PersonalAccount` inherits its balance attribute and message protocol for deposit and withdrawal messages from Accounts in general:

```
(a PersonalAccount) is  (an Account (with balance (≥ 0)))
```

In addition to being an `Account`, a `PersonalAccount` is a `Possession` with owners a set of people:

```
(a PersonalAccount) is
     (a Possession
        (with owners (a Set (withEach element (a Person)))))
```

From the description of a `Possession`, each `PersonalAccount` inherits the owners attribute and the message protocols for ownership.

A PersonalAccount inherits attributes and behavior from *both* the description of an Account *and* the description of a Possession.

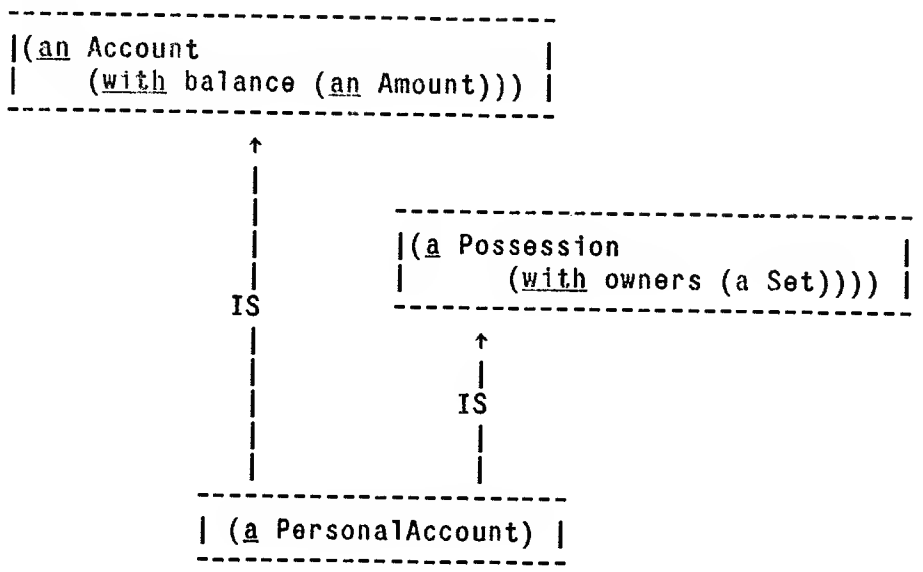

Figure 6-1: Multiple Inheritance

Dealing with the issues raised by the possibility of being a specialization of more than one description has become known as the "Multiple Inheritance Problem". A number of approaches have been developed in the last few years including the following: [Weinreb, Moon 81], [Curry, Baer, Lipkie, Lee 82], [Borning, Ingalls 82], [Bobrow, Stefik 82] and [Borgida, Mylopoulos, Wong 82]. Our approach differs in that it builds on the theory of an underlying description system [Attardi, Simi 81] and in the fact that it is designed for a parallel message passing environment in contrast to the sequential coroutine object-oriented programming languages derived from Simula. Traditional properties of transactions (e.g. the all or nothing property) can be implemented by actors following the appropriate message protocols. These actors are called transaction managers. PersonalAccount actors could be implemented as specializations of transaction managers and thereby acquire the proper message protocol.

If an actor can change its local state, it is called a *serialized* actor. A serialized actor accepts only one message at a time for processing; it will not accept another message for processing until it has dealt with the one which it is processing in at least a preliminary fashion. All messages received when a serialized actor is processing a message are queued in order of arrival until they can be examined.

Below we present *part* of the implementation of PersonalAccounts. This implementation is written out in "parenthesized English" ([Kahn 79], [ROSIE 81], [Hewitt, Attardi, Simi 80], [Lieberman 81a], [Theriault 82]) which is gradually developing into a technical language. The underlined words have special technical meanings.

```
(Define action
  (new PersonalAccount (with (balance of Account) b)
                       (with (owners of Possession) s))
  to have the following implementation:
  (create a new actor with the behavior that
    after acceptance, select one of the following handlers
    for the communication accepted:
      (If it is (query (balance of Account)) request,
         (reply b))
      (If it is (a Withdrawal (with Amount w)) request,
         (select one of the following cases for w:
           (If it is (less than or equal to b),
              (reply (a Completion Report)) as well as
              (become (new PersonalAccount
                           (with (balance of Account) (b - w)))))
           (If it is (greater than b),
              (complain (an Overdraft Complaint)))))
      (If it is (query (owners of Possession)) request,
         (reply s))
      ...
      ...
      ...)
```

At this point we would like to take note of several unusual aspects of the above implementation. By default, commands in the body of a procedure are executed concurrently. For example, in the communication handler for `Withdrawal` requests above, the following two commands are executed concurrently:

```
(reply (a Completion Report))
(become (new PersonalAccount (with Account Balance (b - w))))
```

The principle of maximizing concurrency is fundamental to the design of actor programming languages. It accounts for many of the differences with conventional languages based on communicating sequential processes. Another example of maximizing concurrency is that the processing of messages by a serialized actor can be pipelined. Serialized actors can be pipelined since processing on a subsequent message can commence once the become command has been executed since it designates the actor which will process the subsequent message. For example after accepting a `Withdrawal` message then executing the following become command

```
(become (new PersonalAccount (with Account Balance (b - w))))
```

a `PersonalAccount` can then *concurrently* accept an `Account Balance` query and reply with the Completion Report for the `Withdrawal` request.

An important special case occurs when an actor can never change its local state. Such an actor is called

unserialized and is treated specially so that conceptually it is able to process arbitrarily many messages at the same time. Actors such as the square root function and the text of Lincoln's Gettysburg Address are unserialized.

Another unusual aspect is that there are *no* assignment commands. Effects are implemented by an actor changing its *own local state* using a become command [Hewitt, Attardi, Lieberman 79]. We model change by actors which change their own local state. Our conceptual model of change contrasts with the usual computer science notion in which change is modeled by updating the state components of a universal global state [Milne, Strachey 76]. The absence of the existence of a well defined global state is a fundamental difference between the actor model and classical sequential models of computation (viz. [Turing 37], [Church 41], etc.) *Actor systems can perform nondeterministic computations for which there is no equivalent nondeterministic Turing Machine* [Clinger 81]. The nonequivalence points up the limitations of attempting to model parallel systems as nondeterministic sequential machines [Milne, Strachey 76]. Will Clinger has developed an elegant mathematical theory (called Actor Theory) which accounts for capabilities of actor systems which go beyond those of nondeterministic Turing Machines.

The limitations of the nondeterministic sequential model have practical consequences in terms of the properties we ean *prove* about useful concurrent systems. Within the Actor Model [Hewitt, Attardi, Lieberman 79] we can prove that the implementation of `PersonalAccount` given above will respond to each message which is sent to it. This property (called *guarantee of response for messages sent*) cannot be proved in the nondeterministic choice model. Guarantee of response is our generalization of the notion of "termination" for sequential programming. Some models of concurrency require that a shared checking account be implemented as a "process" which must "terminate". In contrast we desire that the checking account be able to continue to process messages indefinitely far into to the future. Current models of communicating sequential processes which require that every process (e.g. a checking account) terminate are not applicable to Open Systems. Suppose we are attempting to withdraw money from a `PersonalAccount` which we share with others. In the nondeterministic choice model [Hoare 78] the oracle (malicious demon) might always choose to have the `PersonalAccount` accept a message from someone else and *never* choose to have it accept the message we sent. Note that we regard this as an issue of conceptual modeling as opposed to an issue of implementation. Implementors of CSP might choose to implement it in such a way that a `PersonalAccount` written in CSP has the property that it always responds to messages which it is sent. Even so within the nondeterministic choice model, it will not be possible to *prove* that the `PersonalAccount` has the property. The above example demonstrates how modeling indeterminancy in terms of the Actor Model as opposed to using nondeterministic choice (the malicious demon model) has important consequences for the ability to derive results about the behavior of Open Systems.

Concurrency in Open Systems stems from the parallel operation of an incrementally growing number of multiple, independent, communicating sites. Sites can join an Open system in the course of its normal operation--sometimes even affecting the results of computations initiated *before* they joined. *Actor Theory has been designed to accurately model the computational properties of Open Systems.* It is a consequence of the Actor Model that purely functional programming languages based on the lambda calculus *cannot* implement shared accounts in Open Systems. The technique promoted by Strachey and Milne [Milne, Strachey 76] for simulating some kinds of parallelism in the lambda calculus using continuations does not apply to Open Systems. The lambda calculus simulation is sequential whereas Open Systems are inherently parallel. Concurrency in the lambda calculus stems from the concurrent reduction/evaluation of various parts of a *single* lambda expression with an environment which is fixed when the lambda expression is created. In Open Systems independent sites can incrementally spawn ongoing computations so that the environment of a computation is not fixed when a computation is begun.

Therefore we take issue with the common thesis that the primary advantage of applicative programming is referential transparency (i.e. the absence of effects). Rather the primary advantage of applicative programming is the attractive way in which actions and the results of actions can be composed. We [Hewitt, Attardi, Lieberman 79] have extended the applicative style of programming and its elegant proof theory [Scott 72] to a more general programming methodology capable of causing and describing effects (such as updating the balance of a shared account) which cannot be implemented in the lambda calculus for an Open System.

The object-oriented programming languages (e.g. [Birtwistle, Dahl, Myhrhaug, Nygaard 73], [Liskov, Snyder, Atkinson, Schaffert 77], [Shaw, Wulf, London 77], and [Ichbiah 80]) are built out of objects (sometimes called "data abstractions") which are completely separate from the procedures in the language. Similarly the lambda calculus programming languages (e.g. [McCarthy 62], [Landin 65], [Friedman, Wise 76], [Backus 78], and [Steele, Sussman 78]) are built on functions and data structures (viz. lists, arrays, etc.) which are separate. SmallTalk [Ingalls 78] is somewhat a special case since it simplified Simula by leaving out the procedures entirely, i.e. it has only classes. The Simula-like languages provide effective support for coroutines but not for concurrency. In contrast the Actor Model is designed to aid in conceptual modeling of shared objects in a highly parallel open systems. Actors serve to provide a unified conceptual basis for functions, data structures, classes, suspensions, futures, procedure invocations, exception handlers, objects, procedures, processes, etc. in all of the above programming languages [Baker, Hewitt 77], [Lieberman 81b], [Hewitt, Attardi, Lieberman 79]. For example sending a request communication generalizes the traditional procedure invocation mechanism which requires that control return to the point of invocation. A request communication contains the mail address of a customer to which the response to the request should be sent as well as the message specifying the task to be performed. In this way, exception handlers [Liskov, Snyder, Atkinson, Schaffert 77], [Ichbiah 80] and co-routines [Birtwistle, Dahl, Myhrhaug, Nygaard 73] are

conveniently unified with other more general control structures. The Actor Model unifies the conceptual basis of the lambda calculus and the object-oriented schools of programming languages--being mathematically defined, it is independent of all programming languages.

We conjecture that actors (unlike lambda expressions, etc.) are the universal objects of concurrent systems and that they can serve as a efficient interface between the hardware and software. Actors provide an absolute conceptual interface between the software and hardware of parallel computer systems. The function of the hardware is to efficiently implement the primitive actors and the ability to communicate in parallel. Software systems in turn can be implemented in terms of actors completely independently of the hardware configuration. A system consisting of multiple processors -- called the APIARY -- is being developed to use the inherent parallelism of actor systems to increase the speed of computation [Hewitt 80].

7. Hypothesis Formation

A Sprite is an actor attached to a description in a description network to process *disseminated* messages [Kornfeld, Hewitt 81], [Kornfeld 82], [Barber 82]. Sprites are used in problem solving to reason about hypotheses and goals. Each Sprite has access to the material disseminated in a single description network -- which can be distributed over many computers. The Principle of Commutativity [Kornfeld, Hewitt 81] is fundamental to the Scientific Community Metaphor which provides the rationale for sprites. By this principle, a sprite S that is applicable to a message M will read the message regardless of whether the sprite is created first and then the message M is disseminated or the message M is disseminated and then S created. The Principle of Commutativity reflects the norm of scientific communities that a scientist should read the literature relevant to the problem it is working on regardless of whether the literature was published before it commenced working on the problem, or the literature was published afterward.

Hypothesis formation is an important reasoning activity that can be performed by sprites. For example, from the hypothesis that Mike uses his credit card properly in 1955, we would like to be able to hypothesize that Mike uses his credit card properly in 1956. For example reasoning about Mike's use of his credit card can easily be performed by the following Sprite which behaves as follows: when an hypothesis that Mike uses his credit properly in 1955 is disseminated then disseminate the hypothesis that he uses it properly in 1956.

```
(when (hypothesis (USE-PROPERLY MIKE 1955) do
      (hypothesize (USE-PROPERLY MIKE 1956))
```

Figure 7-1: A Simple Sprite

Note that the reasoning used by the above Sprite is not isomorphic with the first order logic inference rules for the following implication:

(USE-PROPERLY MIKE 1955) implies (USE-PROPERLY MIKE 1956)

The bug in the above implication manifests itself under the circumstance in which Mike doesn't use his credit card properly in 1956 even though he did use it properly in 1955. Using the rules of first order logic from (not (USE-PROPERLY MIKE 1956)) and the above implication we can logically infer (not (USE-PROPERLY MIKE 1955)) which is a mistake.

The above example illustrates the Contrapositive Bug in first order logic. The Contrapositive Bug is the inability to do hypothesis formation in a first order logic theory. It occurs whenever we wish to infer an hypothesis H2 from an hypothesis H1 recognizing full well that the negation of H2 does not move us to infer the negation of H1.

We will demonstrate that the Contrapositive Bug is a consequence of the truth-theoretic [Tarski 44]) semantics that underlie first order logic. In truth-theoretic semantics, the meaning of a sentence is determined by the models which make it true. For example the disjunction of two sentences is true exactly when neither of its disjuncts is false. Our goal is to prove that if the hypothesis (USE-PROPERLY MIKE 1956) can be validly derived from (USE-PROPERLY MIKE 1955) in first order logic from the sentences S, then (not (USE-PROPERLY MIKE 1956)) can be validly derived from (not (USE-PROPERLY MIKE 1955)). By the Completeness and Deduction Theorems for first order logic, a deduction is valid only if it is true in every model. It follows that in every model of S, (USE-PROPERLY MIKE 1956) is true if (USE-PROPERLY MIKE 1955) is true. One of the fundamental tenets of truth-theoretic semantics is that in a given model, a sentence is either true or false. Therefore (USE-PROPERLY MIKE 1955) must be false in every model of S in which (USE-PROPERLY MIKE 1956) is false. The above example shows how the Contrapositive Bug is a consequence of the truth-theoretic semantics on which first order logic is based.

Other limitations of truth-theoretic semantics as a foundation for reasoning are discussed in the sections below.

8. Due Process

Due Process is the problem solving method of gathering evidence in parallel from parties affected by an issue, weighing the evidence, and then making a decision. *Understanding Due Process Reasoning is central to modeling the reasoning processes that go on in Open Systems.*

In this section we consider a concrete example of Due Process reasoning. Note that the hypothesis that Mike didn't use his credit card properly before 1984 is evidence for the hypothesis that he will not use his

credit card properly in 1984. This reasoning is expressed by the sprite below:

```
(when (hypothesis ((not (USE-PROPERLY MIKE y))
                   and (y PRECEDES 1984)))) do
     (hypothesize (not (USE-PROPERLY MIKE 1984)))
```

Using Due Process reasoning, a goal cannot be established by looking for evidence *only* in favor of the goal. In order to decide whether or not Mike uses his credit card properly in 1984, proponent activities must be established to gather evidence in favor and skeptics to gather evidence against [Kornfeld, Hewitt 81]. The resources devoted to gathering evidence should be appropriate for importance and urgency of the issue being addressed. The evidence is then weighed and if possible a decision is made. For example suppose that we have information that Mike did not use his credit card properly in 1975 but that he did use it properly in both 1982 and 1983. Then the Sprite given above together with the following:

```
(when (hypothesis (USE-PROPERLY MIKE 1983) do
      (hypothesize (USE-PROPERLY MIKE 1984))
```

can be invoked in parallel to produce evidence both for and against the hypothesis that Mike uses his card properly in 1984. Having used it properly in 1982 is evidence in favor of using it properly in 1984 whereas not having used it properly in 1975 is evidence against. Due Process can be implemented by Sprites which have been programmed to weigh conflicting evidence. They can use the hypothesis that a card will be used properly in a year if it has been used properly in recent previous years, even though it was used improperly in the distant past.

The following first order sentences which attempt to summarize the situation we are describing are inconsistent:

```
(USE-PROPERLY MIKE 1982)
(USE-PROPERLY MIKE 1983)
((not (USE-PROPERLY MIKE y)) and (y PRECEDES 1984)) implies
                  (not (USE-PROPERLY MIKE 1984)))
(not (USE-PROPERLY MIKE 1975)
(USE-PROPERLY MIKE 1983) implies (USE-PROPERLY MIKE 1984)
(1975 PRECEDES 1984)
```

Figure 8-1: An Inconsistent Description

By the rules of first order logic, from the above inconsistency any arbitrary conclusion can be drawn no matter how ridiculous. The above example illustrates the Inconsistency Bug in first order logic. The Inconsistency Bug in the inability of a first order logic theory to deal rationally with inconsistent descriptions. It occurs whenever an inconsistent set of axioms is used to derive a theorem which is unsound because the

proof depends in an essential way on the contradictions in the axioms. Like the Contrapositive Bug, the Inconsistency Bug is a consequence of the truth-theoretic semantics that underlie first order logic. Truth-theoretic semantics takes the meaning of a set of sentences to be the set of all models which satisfy the sentences. An inconsistent set of sentences is meaningless because there are no models which satisfy the set.

The history of physics is replete with examples mathematical models which turned out to be contradictory by the derivation of a paradox deep in the middle of a calculation. Even mathematics is not immune to the phenomena of theorems being lost after the underlying theory has been shown to be contradictory. The proofs of such theorems depended on a contradictory axiomatization in some fundamental way [Lakatos 76] and thus the theorems had not been solidly established even though they had been proved. *We conjecture that any set of axioms that purports to describe the expert knowledge of a complicated, real system (viz. human kidneys, the Japanese economy, the US Supreme Court, etc.) will be inconsistent.* From this we conclude that the Inconsistency Bug poses an important problem to the use of first order logic reasoning [Hewitt 75], [diSessa 77], [Minsky 75].

The existence of bugs has not gone unnoticed by advocates of the use of first order logic for the mechanization of reasoning. Some advocates of first order logic concerned with the problems discussed above are attempting to use *multiple* first order logical theories and their meta-theories (viz. [McCarthy 80], [Weyhrauch 80], [Hayes 77], [Moore 82], etc.) One of the approaches is called circumscription. Instead of using a simple implication like the one below

```
(USE-PROPERLY MIKE 1983) implies (USE-PROPERLY MIKE 1984)
```

to express hypothesis formation, the implication below is used instead:

```
((USE-PROPERLY MIKE 1983) and ASSUMPTION-1)
        implies (USE-PROPERLY MIKE 1984)
```

The hypothesis ASSUMPTION-1 is to be taken to be true unless its negation can be proved from the other axioms. Ray Reiter pointed out to us that circumscription presently formulated does not deal at all well with the problem of inconsistent evidence. In the example of Due Process presented above, we would like to be able to hypothesize that Mike uses his credit card properly in 1984 because he used it properly in the previous couple of years. Since the description is inconsistent, (not ASSUMPTION-1) is provable and therefore (USE-PROPERLY MIKE 1984) cannot be inferred from (USE-PROPERLY MIKE 1983).

The development of logic systems to cope with the problems discussed above is currently in a state of rapid flux. Other researchers are investigating the use of higher order logics, modal logics, intuitionistic logics, etc. None of the above modifications to logic address what we consider to be the fundamental limitation of logic:

Mathematical reasoning [De Millo, Lipton, Perlis 79] as well as reasoning about the world is a social process which inherently requires communication among the parties involved. None of the above extensions to logic make provision for the kind of communication required. *We conjecture that it is impossible to implement Due Process Reasoning for Open Systems in first order logic because it lacks the necessary communication capabilities. Due Process inherently involves being open to outside communications, even those put forth by parties which joined the Open System after the issue being acted upon was first posed.*

Due Process is central to the operation of most community decision making (viz. trial courts, legislatures, appeal courts, regulatory agencies, scientific communities, etc). We must address the problem of making computer systems apply Due Process with something approaching the subtlety and sophistication of human communities. Achieving this goal will require further advances in *anthropomorphic programming* which is the design of computer systems on the basis of the same principles used by human communities. *Anthropomorphic programming is the soundest programming methodology currently available and is also the one which shows the greatest promise for future growth.* [Hewitt 69], [Kay, Goldberg 77], [Hewitt 77], [Kornfeld, Hewitt 81] An excellent discussion of many of the issues addressed by anthropomorphic programming is contained in [Booth, Gentleman 82]. Anthropomorphic programming is a powerful tool with which to further the implementation of Due Process Reasoning.

9. Message Passing Semantics

Message Passing Semantics takes a different perspective on the meaning of a sentence from that of truth-theoretic semantics. In truth-theoretic semantics, the meaning of a sentence is determined by the models which make it true. For example the conjunction of two sentences is true exactly when both of its conjuncts are true. *In contrast Message Passing Semantics takes the meaning of a message to be the effect it has on the subsequent behavior of the system.* In other words the meaning of a message is determined by how it affects the recipients. Each partial meaning of a message is constructed by a recipient in terms of how it is processed (c.f. [Reddy 79]). The meaning of a message is open ended and unfolds indefinitely far into the future as other recipients process the message.

At a deep level, understanding always involves categorization, which is a function of interactional (rather than inherent) properties and the perspective of individual viewpoints. Message Passing Semantics differs radically from truth-theoretic semantics which assumes that it is possible to give an account of truth in itself, free of interactional issues, and that the theory of meaning will be based on such a theory of truth [Lakoff, Johnson 80].

An important limitation of truth-theoretic semantics is that although it accounts for some reasoning about hypotheses, it does not account for goals. Goals are not the kind of object that can meaningfully be said to be

either true or false. The distinction between *hypothesis invoked* (i.e. sprites of the form (when (hypothesis ...) ...) and *goal invoked* (i.e. sprites of the form (when (goal ...) ...) reasoning is central Due Process Reasoning. Recall the following sprite considered earlier which performs hypothesis formation.

```
(when (hypothesis (USE-PROPERLY MIKE 1983)) do
      (hypothesize (USE-PROPERLY MIKE 1984))
```

Figure 9-1: An Hypothesis-invoked Sprite

From the above sprite we can *derive* the following:

```
(when (goal (USE-PROPERLY MIKE 1984)) do
      (show (USE-PROPERLY MIKE 1983)))
```

Figure 9-2: A Goal-invoked Sprite

When the above sprite is invoked with the goal that Mike uses his card properly in 1984, it disseminates the subgoal that he uses his card properly in 1983. The sprite shown below shows how the goal-invoked sprite can be derived from the hypothesis-invoked sprite:

```
(when
      (hypothesis
                (when (hypothesis antecedent) do
                      (hypothesize consequent)))
 do
      (hypothesize
                (when (goal consequent) do
                      (show antecedent))))
```

Making the distinction between hypothesis-invoked and goal-invoked reasoning as a key component of a language for problem solving was proposed by [Hewitt 69] and implemented by [Sussman, Winograd, Charniak 70] which built on earlier work by Minsky and Papert as well as Newell, Shaw, and Simon [Newell 62]. *The separation of hypotheses and goals and explicitly reasoning about both is an important advantage of process-based reasoning over truth-theoretic based reasoning* (cf. [Hewitt 75], [de Kleer, Doyle, Steele, Sussman 77]). Winograd made excellent use of this work [Winograd 71] and further developed these ideas [Winograd 80]. The rules for sprites [Kornfeld, Hewitt 81] extend this work to Open Systems.

Sprites can perform logical inferences in contexts where this is appropriate. For example the following sprite

```
(when (hypothesis (and sentence1 sentence2)) do
      (hypothesize sentence1) as well as
      (hypothesize sentence2))
```

will concurrently hypothesize individual sentences (call them `sentence1` and `sentence2`) whenever the conjunction (`and sentence1 sentence2`) is hypothesized.

One of the most challenging problems in the conceptual modeling of Open Systems is sorting out the relationship between *doing* and *describing*. The distinction between doing and describing is well illustrated by office work in large corporations. Policies and procedures manuals, memos, guidelines, etc. provide a *description* whereas the office workers themselves *perform* the work as practical action [Suchman 79]. Currently the discrepancies between the available descriptions and the realities of the actual work are huge [Wynn 79]. Modern office work is learned mainly by apprenticeship.

The distinction between *doing* and *describing* is different from the usual distinction made in the Artificial Intelligence literature [McCarthy, Hayes 69] [Hayes 77] between the *epistemological adequacy* of a system (its *accuracy* with respect to truth-theoretic semantics [Tarski 44]) and the *heuristic adequacy* (the *efficiency* of its inferential procedures in proving theorems). Thus the distinction between epistemological adequacy and heuristic adequacy is founded on the basis of truth-theoretic semantics. A simple example can help clarify the distinction. An epistemologically adequate theory of bicycle riding gives an accurate description of the physics of how a bicycle works. An heuristically adequate system can derive theorems in the theory of bicycle riding fast enough for some purpose. Both kinds of adequacy are concerned with the *description* of bicycle riding: the former with its accuracy; the later with the efficiency with which the description can be used to answer questions. Neither kind of adequacy *accomplishes* riding a bicycle from Boston to Lexington. That is, describing a future 1987 bicycle ride from Boston to Lexington in arbitrarily fine detail will not cause it to actually happen. *Message Passing Semantics deals coherently with both doing and describing whereas truth-theoretic semantics only addresses some of the issues of describing.*

10. Self-Reference, Self-Knowledge, and Self-Development

Self-reference, self-knowledge, and self-development will be important capabilities for the effective utilization of Open Systems. Therefore understanding these capabilities is central to the conceptual modeling of Open Systems. Unfortunately the analysis of the communication semantics of these capabilities is still quite rudimentary. However we feel that the prospects for deeper analysis are excellent. *Note that both negotiation and Due Process are inherently self-referential activities in which reference and appeal to the nature of the ongoing process is often made from within.*

Many of the advantages of self-reference stem from the ability to analyze the meaning of past communications. Such an analysis can indicate that the system is in a rut or alternatively that it is making good progress toward some goal. Analysis of recent communications can provide valuable information as to how certain negotiations are proceeding. Historical analysis provides leverage by giving a subsystem a handle on the semantics of the communications between itself and the external environment. In general, knowledge of the nature of this interaction cannot be derived solely from general theories independently of the historical context.

Of course the ability of a subsystem to abstractly analyze its own mechanisms is also important. Indeed this ability is essential in order to effectively support users in their problem solving activities. When a subsystem runs into difficulty attempting to carry out a task, it needs extensive knowledge of what it is doing and why in order to interact effectively with others to overcome the difficulties.

Additional leverage is provided by the capability for a subsystem to have explicit knowledge of its own goals. Such knowledge enables a system to relate its goals to one another so that in some cases it can detect partial conflicts and overlaps. In addition it can aid in focusing effort by providing information which can potentially be used to help judge whether or not it is spreading its resources too thin.

11. Conclusion

In this paper we have discussed a number of issues we are addressing in the conceptual modeling of Open Systems. Open Systems are distributed, highly parallel, incrementally evolving, computer systems that are in continuous operation always capable of further growth. The actors in Open Systems must cope with incomplete knowledge and power. In order to function more effectively, the need to be able to negotiate with each other, perform problem solving using Due Process reasoning, and have a good deal of self-knowledge.

Together with our colleagues, we have implemented many *separate* systems to deal with different aspects of the issues involved: Act 1 [Lieberman 81a], Omega [Barber 82], Ether [Kornfeld 82], SBA [Byrd, Smith, de Jong 82], etc. Currently the Message Passing Semantics Group is constructing a system which *unifies* these technologies to support the conceptual modeling of Open Systems. Message Passing Semantics is in an active research phase. The examples presented in this paper represent our current thoughts on what is needed, rather than our final views. Clearly a great deal of study still needs to be performed on difficult issues in the conceptual modeling of Open Systems which have been only touched upon in this paper (such as negotiation, transaction management, problem solving, change, and self-knowledge).

12. Acknowledgments

We would like to thank Michael Brodie for many long discussions concerning conceptual modeling. We would also like to thank the other participants of the Conceptual Modeling Workshop for sharpening our ideas on this subject. A preliminary version of this paper is being published in a volume on the workshop [Conceptual Modeling 82]. Subsequent to the workshop conversations with several of our colleagues provided additional valuable insights which have been incorporated: Discussions with David Israel, John McCarthy, and Ray Reiter helped us to sharpen our critique of first order logic. Fanya Montalvo helped develop the section on self knowledge and suggested how to make the paper more coherent. A discussion with Allen Newell suggested that Due Process Reasoning is even more fundamental than first supposed and perhaps can be used to derive what he calls "weak methods". Charles Smith made valuable suggestions on how to better focus the beginning of the paper. Jerry Barber suggested an improved title. A conversation with John Wheeler established some fascinating connections with developments in modern physics which will be a topic in a forthcoming paper. Discussions with Bob Moore helped us to sharpen the distinction between our approach to the semantics of communication and the truth theoretic approach. For many years the first author has benefitted from discussions with Richard Weyhrauch who has a deep understanding of the strengths and weaknesses of first order logic. Randy Davis pointed out several weak points in our presentation. Jonathan Amsterdam, Mike Brady, Toni Cohen, Bob Filman, Kenneth Kahn, JCR Licklider, Dave McDonald, and Daniel Weld provided valuable comments. Very preliminary versions of some of the ideas in this paper benefitted from being critiqued by the participants at the the Distributed Artificial Intelligence Workshop organized by Lee Erman which was held at the USC Conference Center in June 1982.

Much of the work underlying our ideas was conducted by members of the Message Passing Semantics group at MIT. We especially would like to thank Giuseppe Attardi, Jerry Barber, Bill Kornfeld, Henry Lieberman, Dan Theriault, and Maria Simi. The development of the Actor Model has benefited from extensive interaction with the work of Jack Dennis, Dan Friedman, Bert Halstead, Tony Hoare, Gilles Kahn, Dave MacQueen, Robin Milner, Gordon Plotkin, Steve Ward, David Wise over the past half decade. The work on Simula and its successors SmallTalk, CLU, Alphard, etc. has profoundly influenced our work. We are particularly grateful to Alan Kay, Peter Deutsch, and the other members of the Learning Research group for interactions and useful suggestions.

This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Major support for the research reported in this paper was provided by the System Development Foundation. Major support for other related work in the Artificial Intelligence Laboratory is provided, in part, by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505. We would like to thank Charles Smith, Mike Brady, and Patrick Winston for their support and encouragement.

References

[Attardi, Simi 81]
Attardi, G. and Simi, M.
Semantics of Inheritance and Attributions in the Description System Omega.
In *Proceedings of IJCAI 81*. IJCAI, Vancouver, B. C., Canada, August, 1981.

[Backus 78]
Backus, J.
Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
Communications of the ACM 21(8):613-641, August, 1978.

[Baker, Hewitt 77]
Baker, H. and Hewitt, C.
The Incremental Garbage Collection of Processes.
In *Conference Record of the Conference on AI and Programming Languages*. ACM, Rochester, New York, August, 1977.

[Barber 82]
Barber, G. R.
Office Semantics.
PhD thesis, Massachusetts Institute of Technology, 1982.

[Birtwistle, Dahl, Myhrhaug, Nygaard 73]
Birtwistle, G. M., Dahl, O-J., Myhrhaug, B., Nygaard, K.
Simula Begin.
Van Nostrand Reinhold, New York, 1973.

[Bobrow, Stefik 82]
Bobrow, D. G., Stefik, M. J.
Loops: An Object Oriented Programming System for Interlisp.
Draft, Xerox PARK, 1982.

[Booth, Gentleman 82]
Booth, K. S., Gentleman, W. M.
Anthropomorphic Programming.
In *Conference on Language Issue in Large Scale Computing*. Lawrence Livermore Laboratory, March, 1982.

[Borgida, Mylopoulos, Wong 82]
Borgida, A., Mylopoulos, J. L., Wong, H. K. T.
Generalization as a Basis for Software Specification.
In Brodie, M. L., Mylopoulos, J. L., Schmidt, J. W., editor, *Perspectives on Conceptual Modeling*. Springer-Verlag, 1982.

[Borning, Ingalls 82]
Borning, A. H., Ingalls, D. H.
Multiple Inheritance in Smalltalk-80.
In *Proceedings of the National Conference on Artificial Intelligence*. AAAI, August, 1982.

[Brady 82]
Brady, M.
Private communications

[Byrd, Smith, de Jong 82]
Byrd, R. J., Smith, S. E., de Jong, S. P.
An Actor-Based Programming System.
In *Conference on Office Information Systems.* ACM SIGOA, June, 1982.

[Church 41]
Church, A.
The Calculi of Lambda-Conversion.
In *Annals of Mathematics Studies Number 6.* Princeton University Press, 1941.

[Clinger 81]
Clinger, W. D.
Foundations of Actor Semantics.
AI-TR- 633, MIT Artificial Intelligence Laboratory, May, 1981.

[Conceptual Modeling 82]
Brodie, M. L., Mylopoulos, J. L., Schmidt, J. W.
Perspectives on Conceptual Modeling.
Springer-Verlag, N.Y., 1982.

[Curry, Baer, Lipkie, Lee 82]
Curry, G., Baer, L., Lipkie, D., Lee, B.
Traits: An Approach to Multiple-Inheritance Subclassing.
In *Conference on Office Information Systems.* ACM SIGOA, June, 1982.

[Davis, Smith 81]
Davis, R., Smith, R.
Negotiation as a Metaphor for Distributed Problem Solving.
Memo 624, MIT AI Laboratory, May, 1981.

[de Jong 80]
de Jong S. P.
The System for Business Automation(SBA): A Unified Application Development System.
In *Proceedings of the 1980 IFIP Congress.* IFIP, Tokyo, 1980.

[de Kleer, Doyle, Steele, Sussman 77]
de Kleer, J., Doyle, J., Steele, G. L., and Sussman, G. J.
AMORD: Explicit Control of Reasoning.
In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages.* SIGART, Rochester, N.Y., August, 1977.

[De Millo, Lipton, Perlis 79]
De Millo, R., Lipton, R., and Perlis, A.
Social Processes and Proofs of Theorems.
Communications of the ACM 22(5):271-280, May, 1979.

[diSessa 77]
diSessa, A.
On Learnable Representations of Knowledge: A Meaning for the Computational Metaphor.
AI Memo 441, MIT, September, 1977.

[Friedman, Wise 76]
Friedman, D. P., Wise, D. S.
The Impact of Applicative Programming on Multiprocessing.
In *Proceedings of the International Conference on Parallel Processing*, pages 263-272. ACM, 1976.

[Hayes 77]
Hayes, P. J.
In Defense of Logic.
In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 559-565. Cambridge, Ma, 1977.

[Hewitt 69]
Hewitt C. E.
PLANNER: A Language for Proving Theorems in Robots.
In *Proceedings of IJCAI-69.* IJCAI, Washington D. C., May, 1969.

[Hewitt 75]
Hewitt, C.
How To Use What You Know.
In *Proceedings of IJCAI 75.* IJCAI, Tiblisi, Georgia, USSR, August, 1975.

[Hewitt 77]
Hewitt C.
Viewing Control Structures as Patterns of Passing Messages.
Artificial Intelligence 8:323-364, 1977.

[Hewitt 80]
Hewitt C. E.
The Apiary Network Architecture for Knowledgeable Systems.
In *Conference Record of the 1980 Lisp Conference.* Stanford University, Stanford, California, August, 1980.

[Hewitt, Attardi, Lieberman 79]
Hewitt C., Attardi G., and Lieberman H.
Specifying and Proving Properties of Guardians for Distributed Systems.
In *Proceedings of the Conference on Semantics of Concurrent Computation.* INRIA, Evian, France, July, 1979.

[Hewitt, Attardi, Simi 80]
Hewitt, C., Attardi, G., and Simi, M.
Knowledge Embedding with a Description System.
In *Proceedings of the First National Annual Conference on Artificial Intelligence.* American Association for Artificial Intelligence, August, 1980.

[Hewitt, Baker 77]
Hewitt, C. and Baker, H.
Laws for Communicating Parallel Processes.
In *1977 IFIP Congress Proceedings.* IFIP, 1977.

[Hoare 78]
Hoare, C. A. R.
Communicating Sequential Processes.
CACM 21(8):666-677, August, 1978.

[Ichbiah 80]
Ichbiah, J. D.
Reference Manual for the Ada Programming Language.
November 1980 edition, United States Department of Defense, 1980.

[Ingalls 78]
Ingalls D.
The SmallTalk-76 Programming System, Design and Implementation.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages.*
ACM, Tucson, Arizona, January, 1978.

[Kahn 79]
Kahn, K. M.
Creation of Computer Animation from Story Descriptions.
PhD thesis, Massachusetts Institute of Technology, 1979.

[Kay, Goldberg 77]
Kay A., Goldberg A.
Personal Dynamic Media.
IEEE 10(3), March, 1977.

[Kerns 80]
Kerns, B.
Towards a Better Definition of Transactions.
Technical Report, M.I.T. A.I. Laboratory, December, 1980.

[Kornfeld 82]
Kornfeld, W.
Concepts in Parallel Problem Solving.
PhD thesis, Massachusetts Institute of Technology, 1982.

[Kornfeld, Hewitt 81]
Kornfeld, W. A. and Hewitt, C.
The Scientific Community Metaphor.
IEEE Transactions on Systems, Man, and Cybernetics SMC-11(1), January, 1981.

[Lakatos 76]
Lakatos, Imre.
Proofs and Refutations: The Logic of Mathematical Discovery.
Cambridge University Press, 1976.

[Lakoff, Johnson 80]
Lakoff, G., Johnson, M.
Metaphors We Live By.
The University of Chicago Press, 1980.

[Landin 65]
Landin, P.
A Correspondence Between ALGOL 60 and Church's Lambda Notation.
Communication of the ACM 8(2), February, 1965.

[Lenat 82]
Lenat, D. B.
The Nature of Heuristics.
Artificial Intelligence (), , 1982.

[Lieberman 81a]
Lieberman, H.
A Preview of Act-1.
A.I. Memo 625, MIT Artificial Intelligence Laboratory, 1981.

[Lieberman 81b]
Lieberman, H.
Thinking About Lots of Things At Once Without Getting Confused: Parallelism in Act-1.
A.I. Memo 626, MIT Artificial Intelligence Laboratory, 1981.

[Liskov, Snyder, Atkinson, Schaffert 77]
Liskov B., Snyder A., Atkinson R., and Schaffert C.
Abstraction Mechanism in CLU.
Communications of the ACM 20(8), August, 1977.

[McCarthy 62]
McCarthy, John.
LISP 1.5 Programmer's Manual.
The MIT Press, Cambridge, Ma., 1962.

[McCarthy 80]
McCarthy, J.
Circumscription - A Form of Non-Monotonic Reasoning.
Artificial Intelligence 13(1,2):27-39, April, 1980.

[McCarthy, Hayes 69]
McCarthy, J. and Hayes, P. J.
Some Philosophical Problems from the Standpoint of Artificial Intelligence.
In *Machine Intelligence 4*, pages 463-502. Edinburgh University Press, 1969.

[Milne, Strachey 76]
Milne, R. and Strachey, C.
A Theory of Programming Languages.
John Wiley & Sons, New York, 1976.

[Minsky 75]
Minsky, M.
A Framework for Representing Knowledge.
In Winston, P., editor, *The Psychology of Computer Vision.* McGraw-Hill, New York, 1975.

[Moore 82]
Moore, R. C.
The Role of Logic in Knowledge Representation and Commonsense Reasoning.
In *Proceedings of the National Conference on Artificial Intelligence.* AAAI, August, 1982.

[Newell 62]
Newell, A.
Some Problems of Basic Organization in Problem-Solving Programs.
Memorandum RM-3283-PR, Rand, December, 1962.

[Reddy 79]
Reddy, M.
The Conduit Metaphor.
In Ortony, A., editor, *Metaphor and Thought.* Cambridge University Press, 1979.

[Reiter 82]
Reiter, R.
Towards a Logical Reconstruction of Relational Database Theory.
In Brodie, M. L., Mylopoulos, J. L., Schmidt, J. W., editor, *Perspectives on Conceptual Modeling.* Springer-Verlag, 1982.

[ROSIE 81]
Hayes-Roth, F., Gorlin, D., Rosenschein, S., Sowizral, H., and Waterman, D.
Rationale and Motivation for ROSIE.
Technical Report N-1648-ARPA, RAND, November, 1981.

[Scott 72]
Scott, D. S.
Lattice Theoretic Models for Various Type-free Calculi.
In *Proceedings 4th International Congress in Logic, Methodology and the Philosophy of Science.* , Bucharest, Hungary, 1972.

[Shaw, Wulf, London 77]
Shaw, M., Wulf, W. A., London, R. L.
Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators.
Communications of the ACM 20(8), August, 1977.

[Steele, Sussman 78]
Steele G. L. Jr., Sussman, G. J.
The Revised Report on SCHEME: A Dialect of LISP.
AI Memo 452, MIT, January, 1978.

[Suchman 79]
Suchman, L.
Office Procedures as Practical Action: A Case Study.
Technical Report, XEROX PARC, September, 1979.

[Sussman, Winograd, Charniak 70]
Sussman, G. J., Winograd, T., and Charniak, E.
MICRO-PLANNER Reference Manual.
AI Memo 203, MIT AI Lab, 1970.

[Tarski 44]
Tarski, A.
The Semantic Conception of Truth.
Philosophy and Phenomenological Research 4 :341-375, 1944.

[Theriault 82]
Theriault, D.
A Primer for the Act-1 Language.
A.I. Memo 672, MIT Artificial Intelligence Laboratory, April, 1982.

[Turing 37]
Turing, A. M.
Computability and λ-definability.
Journal of Symbolic Logic 2:153-163, 1937.

[Weinreb, Moon 81]
Weinreb, D. and Moon D.
LISP Machine Manual.
MIT, 1981.

[Weyhrauch 80]
Weyhrauch, R.
Prolegomena to a Theory of Mechanized Formal Reasoning.
Artificial Intelligence 13(1,2):133-172, April, 1980.

[Winograd 71]
Winograd T.
Procedures as a Representation for Data in a Computer Program for Understanding Natural Language.
MAC TR 83, MIT, 1971.

[Winograd 80]
Winograd, T.
Extended Inference Modes in Reasoning.
Artificial Intelligence 13(1,2):5-26, April, 1980.

[Wynn 79]
Wynn, E.
Office Conversation as an Information Medium.
PhD thesis, Department of Anthropology, University of California, Berkeley, 1979.